这是一篇未完成的文章，未完成的原因的包括(1)渲染效果太差(2)看不懂隐式积分。

这篇的内容，基本上就是总结了下面三个参考的内容

https://github.com/YuCrazing/Taichi

[1]FEM Simulation of 3D Deformable Solids: A practitioner's guide to theory, discretization and model reduction.

[2]Dynamic Deformables: Implementation and Production Practicalities

[3]Invertible Finite Elements For Robust Simulation of Large Deformation

代码在finitelement/femcourse中

为了更好地解释形变模型的选择，下面举一个简单的例子。

布料模拟大部分都和衣服有关，因此直接拿衣服来举例子。比如我为了探索出哪种衣服最适合我，因此买了一衣柜的衣服，大小不同，款式各异，材料不一，比如有不易变形的木质衣服，有形变为各项异性的钢筋混凝土衣服，也有柔软的橡胶衣服，而且它们都是由三角形构成的。

接下来需要选一件衣服出门，因此需要出门后有各种意外因素。比如衣服上原来的一个三角形，它原来三个顶点位置是[0,0],[1,0]和[0,1]，那么

$$\frac{1}{\partial \mathbf{X}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}^{-1}$$

现在，三角形由于被人挤一下的影响而形变了，然而衣服作为弹性材料是不喜欢形变的，三角形被改变面积后，肯定会想一些办法再改回去。这是因为组成三角形的材料有一定强度，这个材料被压缩后，产生了能量，三角形想凭借着这股能量恢复到初始形状。

这么说着很容易，但这能量究竟是多大呢？这取决于衣服的材料。更确切地说，我们想让它多大就让它多大，只要根据需要选择合适的衣服就行了。不过，选择衣服的时候有一些标准，需要遵守，不然容易被自己的衣服玩死。

准则一，我们希望衣服在没有形变的时候，没有能量任何能量产生。否则衣服放得好好的，突然自己走起来，肯定会暴露自己的，说好的建国后不准成精啊。

比如我们挑一件Dirchlet的衣服，它的能量计算公式

$$\psi = ||\mathbf{F}||_F^2 = \sum_{i=0}^{2}\sum_{j=0}^{2} f_{ij}^2$$

那么即使这个三角形一直维持三个顶点在相同的位置，那么它的形变梯度为

$$\mathbf{F} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

能量计算为

$$\psi = 1^2 + 0^2 + 0^2 + 1^2 = 2$$

这样衣服就在面积相比与初始状态毫无变化的情况下，产生了能量。这种Dirchlet的衣服如今在市面上已难觅踪迹，原因懂的都懂。

当然，我们可以使用Dirchlet衣服的二维改良版，它的能量公式如下

$$\psi = ||\mathbf{F}||_F^2 - 2$$

注意这个公式和参考[2]的公式(2.6)不同，因为后者是三维的。总之，使用了这个衣服，可以保证衣服在没有变形的情况下的，产生的能量为零。

然而缺点很容易看出来，当衣服收缩的时候，能量就会小于零了。很好，这意味着衣服的质量是负数，也就是这件衣服变成黑洞了，所以这种衣服早就被禁止使用了。为此需要重新定义一个准则。

准则二，由于形变产生的能量是正数或零。

恰好，这里有一种烂大街的衣服，连续签到三天就可以赠送【误】，叫做线弹性模型(linear elasticity)

$$\psi = \mu\varepsilon : \varepsilon + \frac{\lambda}{2} tr^2(\varepsilon)$$

其中varepsilon是strainTensor

$$\varepsilon = (F + F^T)/2 - I$$

假如三角形三个顶点初始为[0,0],[1,0],[0,1]，之和变为[0,0],[x,0],[0,1]，x为任意值，那么应变张量是

$$\varepsilon = \begin{bmatrix} x-1 & 0 \\ 0 & 0 \end{bmatrix} \qquad \psi = \mu(x-1)^2 + \frac{\lambda}{2}(x-1)^2$$

这种线弹性衣服，能量的大小，和形变的二次方大约成比例。当x = 1的时候，能量为零，并且能量不为负数。

这种衣服在前期算是不错的了，就像“村里最好的剑”。简单易用，优点明显，缺点更明显。这里是线弹性的一些档案

- linear elastic，参考[1]的 p16 section 3.2
- 恢复力与形变程度成比例，或者说是线性的，简单友好
- 适于处理小形变，当形变程度太大，由于恢复力与之是线性关系，所以恢复初始形状起来很缓慢
- 物体旋转也会被当作是形变，这是错误的，因为旋转并未造成面积改变

下面的代码来自开源库PositionBasedDynamics库，

```
    const Real trace = epsilon(0, 0) + epsilon(1, 1) + epsilon(2,
2);
    const Real ltrace = lambda*trace;
    sigma = epsilon * 2.0*mu;
    sigma(0, 0) += ltrace;
    sigma(1, 1) += ltrace;
    sigma(2, 2) += ltrace;
    sigma = F * sigma;

    Real psi = 0.0;
    for (unsigned char j = 0; j < 3; j++)
        for (unsigned char k = 0; k < 3; k++)
            psi += epsilon(j, k) * epsilon(j, k);
    psi = mu*psi + static_cast<Real>(0.5)*lambda * trace*trace;
    energy = restVolume * psi;
```

stabel flesh是这么写的

```
Vector24 Cube::_ComputeForces(const Material& material, const
std::vector<Vector3>& verts, const int gpIdx) const
{
    const Matrix3 F = ComputeF(verts, gpIdx);
    Matrix3 U;
    Matrix3 V;
    Vector3 S;
    if (material.PK1NeedsSVD())
        RotationVariantSVD(F, U, V, S);
    const Matrix3 P = material.PK1(F, U, V, S);
    const Matrix38 G = - P * _Bmg[static_cast<unsigned long>
(gpIdx)];
    const Vector24 force = Flatten(G);
    return force;
}
```

这个库非常简洁易用。然而仅仅算出能量是没有用的。对于超弹性材料(hyperelastic materials)，first piola krichhoff stress仅仅和形变梯度有关，并且是让能量对形变梯度求导，这样即可得到

$$\frac{\partial \Psi}{\partial \mathbf{F}} = \mathbf{P}(\mathbf{F}) = \mu(\mathbf{F} + \mathbf{F}^T - 2\mathbf{I}) + \lambda\, tr(\mathbf{F} - \mathbf{I})\mathbf{I}$$

再让P对初始三角的位置的求导，就能计算结点力，

$$\vec{f}(\vec{X}) = \mathbf{div}_{\vec{X}}\mathbf{P}(\vec{X})$$

其中那个大X就是初始未变形的位置。然和就可以根据参考[1]第30页的算1流程写出下面的代码

```
import numpy as np
# 初始化三角形初始位置
node_pos = np.array([[0,0],[1,0],[0,1]])
# 顶点的位置梯度
Ds = np.array([[node_pos[1,0] - node_pos[0,0],node_pos[2,0] -
node_pos[0,0]],
               [node_pos[1,1] - node_pos[0,1],node_pos[2,1] -
node_pos[0,1]]])
# 求逆，用于准备计算形变梯度
minv = np.linalg.inv(Ds)
```

```
# 假设某一时刻，三角形变化到了这样的位置
node_pos = np.array([[0,0],[0.5,0],[0,1]])

time = 0
timeFinal = 100
while(time < timeFinal):
    time += 1
    # 形变梯度中的分子
    Ds_new = np.array([[node_pos[1,0] - node_pos[0,0],node_pos[2,0]
- node_pos[0,0]],
                   [node_pos[1,1] - node_pos[0,1],node_pos[2,1] -
node_pos[0,1]]])
    # 形变梯度
    F = np.dot(Ds_new,minv)
    # 应力，也就是varepsilon
    strain = (F + F.T) * 0.5 - np.identity(2)
    # lame常数
    mu = 1
    # lame常数
    la = 1
    # 线弹性的能量计算公式
    energy = np.inner(strain,strain)[0,0] * mu + la / 2 *
(strain[0,0] + strain[1,1])**2
    #first piola kirchhoff stress
    piola = mu * (F + F.T - 2 * np.identity(2)) + la * (F[0,0] - 1
+ F[1,1] - 1) * np.identity(2)
    # 三角形面积
    area = 0.5
    # 计算力
    H = area * np.dot(piola,Ds.transpose())

    gradC0 = np.array([H[0,0],H[1,0]])
    gradC1 = np.array([H[1,0],H[1,1]])
    gradC2 = np.array([-H[0,0]-H[1,0],-H[1,0]-H[1,1]])

    node_force = np.zeros((3,2))
    #第一个顶点
    node_force[0,:] = gradC0
    #第二个顶点
    node_force[1,:] = gradC1
    #第三个顶点
    node_force[2,:] = gradC2
```

```
    invMass = 1

    dt = 1

    sumGradC = invMass * (gradC0[0]**2 + gradC0[1]**2)
    sumGradC += invMass * (gradC1[0]**2 + gradC1[1]**2)
    sumGradC += invMass * (gradC2[0]**2 + gradC2[1]**2)
    if sumGradC < 1e-10:
        break

    # node_pos[0,:] += dt * energy / sumGradC * invMass * gradC0
    node_pos[1,:] += dt * energy / sumGradC * invMass * gradC1
    node_pos[2,:] += dt * energy / sumGradC * invMass * gradC2
```

注意这个公式也在参考[3]第四节以下面的方式出现

$$\mathbf{G} = \mathbf{P}\mathbf{B}_m$$

G则是节点力，P则是第一Piola Kirchhoff stress。算出来之后，三角形最终的形变梯度为

$$\mathbf{F} = \begin{bmatrix} 1 & 0.5 \\ -0.5 & 1 \end{bmatrix}$$

尽管很容易看出来，三角形此时的面积已经变化，但是依据上面的计算法则，应变和能量是零，也就是线弹性模型认为这种情况下三角形相比于初始状态没有发生任何变化。

emm....线弹性模型开心就好。

接下来换stvk，它需要用到

```
# Greeb Strain，也就是E
E = (np.dot(F,F.T)- np.identity(2)) * 0.5
# st.venant-kirchhoff model
energy = np.inner(E,E)[0,0] * mu + la / 2 * (E[0,0] + E[1,1])**2
#first piola kirchhoff stress
piola = np.dot(F,2 *mu * E + la * (E[0,0] + E[1,1]) *
np.identity(2))
```

那么三角形最终位置和形变梯度为

$$\mathbf{x} = \begin{bmatrix} -0.2692 & 0.6538 & 0.1153 \\ 0.1538 & -0.2307 & 1.0769 \end{bmatrix} \qquad \mathbf{F} = \begin{bmatrix} 0.9230 & 0.3846 \\ -0.3846 & 0.9230 \end{bmatrix}$$

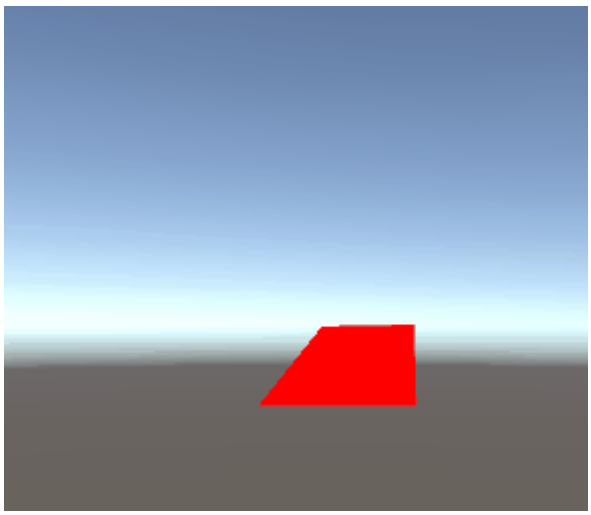
这样最终三角形最终面积仍然是0.5，F也是一个旋转矩阵。角度约为22.6度，也就是0.3948。

接下来是一个非常重要，非常重要，非常重要的准则，重要的事说三遍，

- St. Venant-Kirchhoff model，参考[1]的p18 section 3.3
- 不受刚体运动，即旋转和位移的影响，因为这两种都不算形变，面积没有改变
- 因为我们选择的是基于GreenStrain的模型，算的是形变梯度的二次方，这种模型会忽视面积符号的改变，只求把模型面积的绝对值弄成原来的样子。因此如果面积/体积被改变得小于零，那么改变也仍然会继续下去。详见参考[2]第3节
- 当面积被改变的范围在0~58%的范围内，恢复力会随着形变程度的提高而增强，然而超过这个范围后，恢复力将减弱，因此不适用于大形变

至于第三个特点，为什么是58%，我们可以写个程序验证一下

```
import numpy as np
F = np.array([[1,0],[0,0.4]])
H = np.zeros((100,2,2))
for i in range(100):
    F[1,1] = i / 100
    E = np.dot(F,F.T) - np.identity(2)
    P = np.dot(F,2*E + (E[0,0] + E[1,1])*np.identity(2))
    H[i,:,:] = - P
```



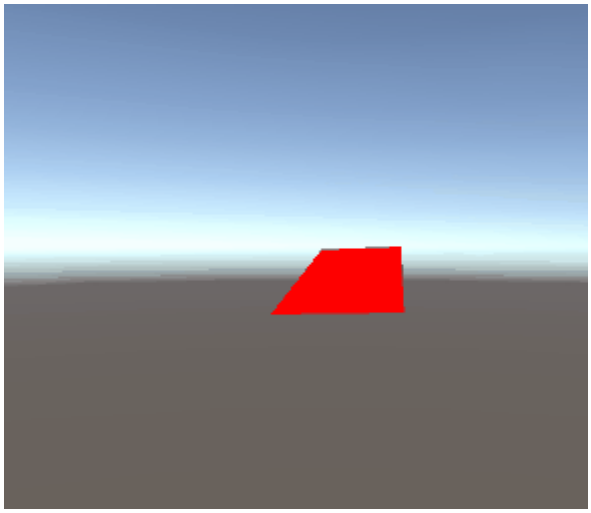
arap 参考[2]的第(3.26)，依然是计算出deformationGradient后，计算出H之前，可以可看参考[2]

```
    U,sigma,Vt = np.linalg.svd(F)
    L = np.identity(2)
```

```
    L[1,1] = np.linalg.det(np.dot(U,Vt))
    detU = np.linalg.det(U)
    detV = np.linalg.det(Vt)
    if detU < 0 and detV > 0:
        U = np.dot(U,L)
    elif detU > 0 and detV < 0:
        Vt = np.dot(Vt,L)
    sigma = np.dot(sigma,L)
    R = np.dot(U,Vt)
    F_R = np.linalg.norm(F - R)**2
    F_R_1 = np.linalg.norm(np.linalg.inv(F) - np.linalg.inv(R))**2
    energy = 0.5 * mu * (F_R + F_R_1)
    piola = mu * (F - R)
```

coratational

```
    U,sigma,Vt = np.linalg.svd(F)
    L = np.identity(2)
    L[1,1] = np.linalg.det(np.dot(U,Vt))
    detU = np.linalg.det(U)
    detV = np.linalg.det(Vt)
    if detU < 0 and detV > 0:
        U = np.dot(U,L)
    elif detU > 0 and detV < 0:
        Vt = np.dot(Vt,L)
    sigma = np.dot(sigma,L)
    R = np.dot(U,Vt)
    S = np.dot(np.linalg.inv(R),F)
    vareps = S - np.identity(2)
    energy = mu * np.inner(vareps,vareps)[0,0] + 0.5 * la *
(vareps[0,0] + vareps[1,1])**2
    RF_I = np.dot(R.T,F) - np.identity(2)
    piola = 2 * mu * (F - R) + la * (RF_I[0,0] + RF_I[1,1]) * R
```



svd永远为正

我们计算出了cauchy stress sigma和pk2，也可见参考[3]如下计算

$$\mathbf{P} = \mathbf{F}\mathbf{S} \qquad \mathbf{P} = J\sigma\mathbf{F}^{-T} \qquad J = det(\mathbf{F})$$

neohookean可以继续如下写参考[2] (4.15)

$$\Psi = \frac{\mu}{2}(||\mathbf{F}||_F^2 - 3) - \mu\log(J) + \frac{\lambda}{2}(\log(J))^2$$

以及piola

$$\mathbf{P} = \mu(\mathbf{F} - \frac{1}{J}\frac{\partial J}{\partial \mathbf{F}}) + \lambda\frac{\log J}{J}\frac{\partial J}{\partial \mathbf{F}}$$

https://github.com/Haotian-Yang/Cloth-Simulation/blob/master/src/dV_membrane_corotational_dq.cpp

https://github.com/JoshWolper/quik-deform/blob/master/TriangleStrainConstraint.cpp

```
    //Fix for inverted elements (thanks to Danny Kaufman)
    double det = S[0]*S[1];

     if(det <= -1e-10)
    {
        if(S[0] < 0) S[0] *= -1;
        if(S[1] < 0) S[1] *= -1;
        if(S[2] < 0) S[2] *= -1;
    }
```

```
    if(U.determinant() <= 0)
    {
        U(0, 2) *= -1;
        U(1, 2) *= -1;
        U(2, 2) *= -1;
    }

    if(W.determinant() <= 0)
    {
        W(0, 2) *= -1;
        W(1, 2) *= -1;
        W(2, 2) *= -1;
    }
```

准则三：当模型仅有纯旋转的形变的时候，也不应该产生能量！

这是因为，衣服中的三角形需要抵抗的是导致它的面积/体积产生变化的力，而和衣服的位置以及旋转角度无关。

因此，我们可以对线弹性这种粗布短褐大肆批判一番了。

PositionBasedDynamics库中的PositionBasedDynamics.cpp第936行开始的代码如下

```
    Eigen::Matrix<Real, 3, 2> H = area * piolaKirchhoffStres *
invRestMat.transpose();
    Vector3r gradC[3];
    for (unsigned char j = 0; j < 3; ++j)
    {
        gradC[0][j] = H(j,0);
        gradC[1][j] = H(j,1);
    }
    gradC[2] = -gradC[0] - gradC[1];


    Real sum_normGradC = invMass0 * gradC[0].squaredNorm();
    sum_normGradC += invMass1 * gradC[1].squaredNorm();
    sum_normGradC += invMass2 * gradC[2].squaredNorm();

    // exit early if required
    if (fabs(sum_normGradC) > eps)
    {
        // compute scaling factor
```

```
        const Real s = energy / sum_normGradC;

        // update positions
        corr0 = -(s*invMass0) * gradC[0];
        corr1 = -(s*invMass1) * gradC[1];
        corr2 = -(s*invMass2) * gradC[2];

        return true;
    }
```

非旋转SVD

```
static void RotationVariantSVD(const Matrix3& F, Matrix3& U,
Matrix3& V, Vector3& S)
{
    const Eigen::JacobiSVD<Matrix3,Eigen::NoQRPreconditioner>
svd(F, Eigen::ComputeFullU | Eigen::ComputeFullV);
    U = svd.matrixU();
    V = svd.matrixV();
    S = svd.singularValues();
    if (U.determinant() < 0.0)
    {
        U.col(0) *= -1.0;
        S(0) *= -1.0;
    }
    if (V.determinant() < 0.0)
    {
        V.col(0) *= -1.0;
        S(0) *= -1.0;
    }
}
```

libigl，将svd转化为rs分解

```
template<typename Mat>
IGL_INLINE void igl::polar_svd3x3(const Mat& A, Mat& R)
{
  // should be caught at compile time, but just to be 150% sure:
  assert(A.rows() == 3 && A.cols() == 3);

  Eigen::Matrix<typename Mat::Scalar, 3, 3> U, Vt;
  Eigen::Matrix<typename Mat::Scalar, 3, 1> S;
  svd3x3(A, U, S, Vt);
  R = U * Vt.transpose();
}
```

以上是一些

比较基础的模型，除此之外，还有一些出没在各种论文中的

比如迪士尼&皮克斯的论文Stable Flesh Simulation，使用了修改后的neo-hookean模型，能量定义如下

$$\Psi_D = \frac{\mu}{2}(I_C - 3) + \frac{\lambda}{2}(J-1)^2$$

pk1定义如下

$$\mathbf{P}_d(\mathbf{F}) = \mu\mathbf{F} + \lambda + \frac{\partial J}{\partial \mathbf{F}}(J-1)$$

按照femsimulation写似乎有问题

```
    logJ = np.log(max(np.linalg.det(F),0.01))
    I_c = F[0,0]**2 + F[1,1]**2
    energy = 0.5 * mu * (I_c - 2) - mu * logJ + 0.5 * la * logJ**2
    F_inv = np.linalg.inv(F)
    F_inv_T = F_inv.T
    piola = mu * (F - mu * F_inv_T) + la * logJ * F_inv_T
```

也可以这么写

```
    pJpF = np.array([[F[1,1],-F[1,0]],[-F[0,1],F[0,0]]])
    J = max(0.01,np.linalg.det(F))
    logJ = np.log(J)
    I_c = F[0,0]**2 + F[1,1]**2
    energy = 0.5 * mu * (I_c - 2) - mu * logJ + 0.5 * la * logJ**2
    piola = mu * (F - 1.0 / J * pJpF) + la * logJ / J * pJpF
```

不过因为neo-hookean的能量定义稍微有一点问题，在仍然有形变的情况下仍然会算出能量等于零，所以我们可以仿照stable neo-hookean中的定义将energy不再作为force的控制的因子，也就是

```
    node_pos[0,:] += dt * gradC0
    node_pos[1,:] += dt * gradC1
    node_pos[2,:] += dt * gradC2
```

看到公式，却不知怎么写成代码？没关系，本专栏的一个重要目的就是搜集开放源码的论文，恰好这篇论文也有源码可看，https://graphics.pixar.com/library/StableElasticity/，其中关于计算pk1和psi的代码如下

```
Scalar StableNeoHookean::Psi(const Matrix3& F, const Matrix3&
/*U*/, const Matrix3& /*V*/, const Vector3& /*S*/) const
{
    const Scalar Ic = F.squaredNorm();
    const Scalar Jminus1 = F.determinant() - 1.0 - _ratio;
    return 0.5 * (_mu * (Ic - 3.0) + _lambda * Jminus1 * Jminus1);
}

static Matrix3 PartialJpartialF(const Matrix3& F)
{
    Matrix3 pJpF;

    pJpF.col(0) = F.col(1).cross(F.col(2));
    pJpF.col(1) = F.col(2).cross(F.col(0));
    pJpF.col(2) = F.col(0).cross(F.col(1));

    return pJpF;
}

Matrix3 StableNeoHookean::PK1(const Matrix3& F, const Matrix3&
/*U*/, const Matrix3& /*V*/, const Vector3& /*S*/) const
{
```

```
    const Matrix3 pJpF = PartialJpartialF(F);
    const Scalar Jminus1 = F.determinant() - 1.0 - _ratio;
    return _mu * F + _lambda * Jminus1 * pJpF;
}
```